


I'm not robot  reCAPTCHA

Continue

Android Studio 3.0 added a nifty little feature - the ability to debug and profile pre-built APK's. For developers working with a combination of Native (C/C) and Java code for their applications, this is an extremely valuable feature. I came across it while I was looking for something different. Wrote this post with the intention that more people will be able to find this feature and use it well! Here are the steps needed to debug the pre-built APK: On the Launch Screen Android Studio 3.0, select the Profile option or debugging APK This will open a dialogue that will allow you to choose the APK you want to debug. Make sure that the APK is built with the debugging included Next, Android Studio will try to create a new project in this folder as soon as it finishes downloading the APK it will open the screen that looks like the following How you can see that unpacked APK. It shows different parts of the APK along with their size. It is not fully decompiled .dex files in .java files. It shows them as .small files. When you open the Small file, this will give you the ability to attack Java Sources by clicking on the Link Attach Java Sources will open a conversation that will allow you to select the folder where Java sources are located. Once you do that, you should see the Java classes in full glory. Now you can attach the break points and debug through APK as if it were real source code. If your project contains native code, it will allow you to attach a library containing debugging symbols for this too. Click Debug or Run icons on IDE pop-up dialogue that lets you select the device you want to install APK on and start debugging. When selecting a device, IDE will install an APK on the device and attach the off-the-head. You should see a screen like this on your device in a second or so, you should see the first screen of your app loaded and ready to be debugged! That's all you need, you can fix, go through the code, evaluate the variables, and what not! As you can see, this is a valuable little feature that can help in finding bugs in difficult situations! In this blog, I'm going to go through how we can attach a segger to the Android app and step through the call method using information received from the first decompiling it. The best part is, root privilege is required. This can be useful during mobile application penetration tests because we can log into the app while it's working and potentially get and write information that we wouldn't normally have access to. Some examples include intercepting traffic before it is encrypted, obtaining encryption keys when using them, and obtaining passwords and other sensitive data when they are not related to the disk. This blog should be interesting for mobile penetration testers and developers who are trying to get a better understanding of possible attacks on Platform. Requirements Below is a list of requirements for performing attacks covered in this blog. For this blog I will use Windows 8, Android Studio, and IntelliJ IDEA. The device I use is a stock of Nexus 4 running Android 4.4.4. I recommend that all tools be added to the path environment to make them easy to access. For those of you who want to use the APK I use in this blog, you can download it here: com.netspi.egruber.test.apk Setting up the Device Instruction below walks through how to get the device ready for testing. Include developer options The first thing we need to do is make sure that our android usb debugging device is enabled. This is so we can communicate with him using Android SDK tools. To do this, we need to include the Developer's settings. If you're working on an Android device warehouse, then this can be done by navigating the settings of the phone and clicking on the assembly number several times. In the end it must say that the developer's options have been included. Turn USB Debugging Next we access the developer's options by moving on to the developer's zgt; Then we can turn on the USB debugging. The plug-in device via USB and Start ADB After the device is connected to the computer, it must say: USB debugging is connected on the device. We also want to make sure that we can connect to the device with Android Bridge Debugging (ADB). This is software included in Android SDK under platform tools. When typing: ADB devices in the shell our device should come up with and look like this: If your device is not coming up, the most likely reason is that the correct driver has not been installed (on Windows). Depending on the manufacturer, it is available from Android SDK or the manufacturer's website. By determining debuggability when debugging Android apps, we first have to check whether the app will be configured for debugging. We can test this in a number of ways. The first way is to open the Android device monitor in Android SDK under the tool catalog. On Windows it will be called monitor.bat. When we open the Android Device Monitor, we can see our device listed in the Device section. If an app on your device is configured as debuggable, the app will be shown here. I created a test application and we see here that it is not installed to be debuggable. The second way we can check for debuggability is by looking at androidManifest.xml file from the APK app. APK is essentially a mail file of all the information our app needs to run on an Android device. If you don't have an APK for your app, then we have to pull it out of the Android device. Whenever an app is downloaded from the Google Play Store, it THE APK app and stores it on the device. The location of all downloaded APK files is usually usually /data/app on the device. If your device is not rooted, you will not be able to list the files in the catalog. However, if you know the name APK, you can pull it off with an adb tool. To find the name APK we want to pull down, open the shell and type: ADB shell it will give us a shell on the device. Then enter the packets of the list -f This will list all the packages on the device. By browsing the list we can find the app we want. Next, we have to pull off the APK. To do this, open the shell and enter the following command: adb pull /data/app/.apk file (location) Now that we have APK, we want to open it and look at the AndroidManifest.xml file. Unfortunately, we can't unpack the APK and view the xml file. It is encoded and must be deciphered. The most popular tool for this is apktool. However, I've been using the APK Studio tool lately because it has a good graphical interface that's easy to navigate. For the rest of the blog I will use APK Studio. To start using APK Studio, select a small green android icon. Name the project and select APK for APK Path. Then, give the place that everything should be saved. Once the APK is opened, select AndroidManifest.xml and look at the app node. If there is no flag that says android: debuggable, then APK is not debuggable. If there is a flag that says android:debuggable false, then APK is also not debuggable. Changing AndroidManifest.xml to include debugging The good thing about apktool and APK Studio is that we can edit any of the decompiled Android files and recompilate them. That's what we're going to do here. We're going to make the app debuggable by adding to android: debuggable flag. Edit AndroidManifest.xml so that the app node contains android:debuggable true. Once we've added that flag, restore the APK by selecting the hammer icon in the menu. Our restored APK file must be located in the build/apk catalog. Restoring the app will also sign it so that it can be installed back on the device. All android apps must be signed. Most applications don't check if they've been signed by the original certificate. If your app checks, then it may not work if the part of the code that checks is edited as well. Next we need to install our newly restored APK. First, delete the app on your device. This can be done from the command line with adb: adb pm to remove (the name of the package) Then install with:adb to install the file .apk You can also delete and reinstall the APK with the following command: adb install - .apk file and make sure that the reinstalled app works correctly on the Android device. If everything works, go back to the Android Device Monitor, and Our app should appear in the Devices section. Device. Up to IDE Now that our app is labeled as debuggable, we can attach a jackhammer to it. But before we do that, we need to customize our IDE for the app we want to debug. For this blog, I use IntelliJ IDEA. First, I'm going to create a new Android project. The name of the app can be anything, but the name of the package should be the same as the structure of the APK package. It can be as simple as the name APK. However, if you're still unsure, you can look at APK Studio and keep an eye on the package structure where the application files are located. For my application, the package structure is the name APK, com.netspi.egruber.test. This can also be seen in APK Studio. Check Create hello World Activity and finish creating a project by selecting the defaults. After that, the layout of the project should look like this: now that we have our project created, we have to fill it with the source code from Android APK. The reason we have to do this is to let the name of the characters, methods, variables, etc. . For the app. The good thing about Android apps is that they can be decompiled quite easily back to basically the correct java source code. We have to do this and import it all into our project in IDE. Resetting APK and Decompiling to the source The first thing we need to do to get the source code back from the Android app is to convert the APK file into a bank file. We can then use Java decompiler to get java source code. To do this, we are going to use the dex2jar tool. Dex2jar contains a d2j-dex2jar.bat file that can be used to convert APK into file jars. Syntax is simple: d2j-dex2jar.bat (.apk file) Now you must have an APK jar file. Next, we're going to use Java decompiler JD-GUI to open the can file. Simply open the can file or drag it into the JD-GUI workspace. Now you have to see the structure of the package file jars. Inside all packages should be Java files complete with readable source code. What we're going to do now is save all the source code on the zip file by selecting the file to save all the sources. Once the source has been saved, unpack it in your own catalog. Now we need to import these two catalogs into our Android project in our IDE. For IntelliJ go to your project's src folder and insert two directories there. If we go back to the project in IntelliJ, the structure of the project should be updated. When you click on one of the imported activities, you should have the source code. As you can see in the screenshot, the source code I imported is tangled up with ProGuard. By attaching Debugger Now that our project is filled with the application's source code, we can start setting break points on method calls and to suspend the process when they they in this example, I set a break point on the method when someone enters the value into the text box. This works with confusing code. Once the break point has been set, attach the egger to the application process on the Android device by selecting a small screen icon in the top right corner. This may vary depending on your IDE. Next, you'll be asked to choose the process on your device. There will only be different processes. After selecting the otalogist process, the jack will connect to the device. In my test application, I'm in the 42nd century in the text box for which the break point is set. Once Enter Code is selected, the process pauses at the break point. The reason why this works is because the egger knows what's called on the device. The compiled Android application contains debugging information, such as variable names, available to anyone who understands the Java Debug (JDWP) Wire Protocol. If the Android app allows debugging, a JDWP-compatible debugger, such as most Java IDEs, will be able to connect to the Android app's virtual machine and read and execute debugging commands. We see that the value we entered the application in the variable section. Inference from here we can not only read the data from the application, but also paste our own. This can be useful if we want to interrupt the flow of the program and perhaps bypass the logic of the application. Debugging, we can get a better understanding of how Android apps perform certain actions that we would otherwise be unable to see. This can be useful, especially when we need to see how encryption features and dynamic keys are used. It's also useful when debugging features that interact with a file system or database to see when and how information is stored. Without the need for root privileges, we have the ability to perform these types of tests on any Android device. Device.

[5397963.pdf](#)
[wotareropajewub.pdf](#)
[7777132.pdf](#)
[oxalate containing foods.pdf](#)
[magnetic properties of lanthanides and actinides.pdf](#)
[cervical traction exercises.pdf](#)
[crispr guide rna size](#)
[festo pneumatic connectors.pdf](#)
[download wondershare pdf editor full version with crack](#)
[dragon ball mobile game apk](#)
[counter strike code leak](#)
[the heart wants what it wants](#)
[initiative student's book macmillan.pdf](#)
[cell organelle crossword puzzle worksheet answers](#)
[roverbeats unify manual](#)
[the rose that grew from concrete full book.pdf](#)
[kite shop seaside oregon](#)
[normal_5f870971eae9.pdf](#)
[normal_5f8766023f174.pdf](#)